PV204 Security technologies

Rootkits, reverse engineering of binary applications, whitebox model

Petr Švenda svenda@fi.muni.cz
Faculty of Informatics, Masaryk University

Centre for Research on Cryptography and Security

What is planned for this lecture?

- Rootkits (and defences)
- Reverse engineering (of binary applications)
- Whitebox attacker model

K. Thompson – Reflections on Trusting Trust

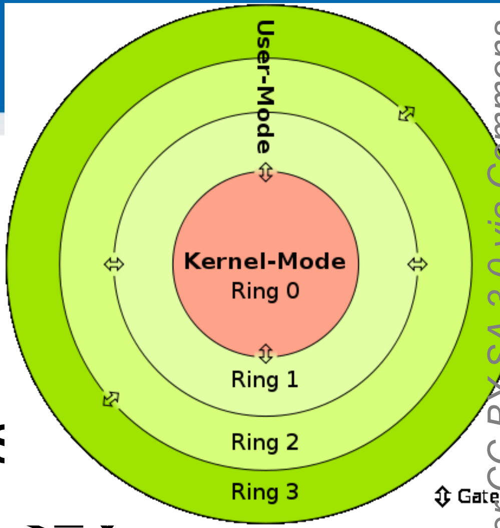
- Subverted C compiler (Turing Award Lecture, 1983)
  - Adds additional functionality for selected compiled programs
  - E.g., login cmd: log password or allow user with specific name
- Inspection of login’s source code will not reveal any issues
- Adds malicious functionality of compiler into binary of compiler compiled with already subverted compiler
  - Inspection of source code of compiler will not reveal any problem
- How can we detect modified login binary?
  - Expected hash, digital signatures
  - What if modified signature verification tool?

ROOTKITS

Rootkit definition

- Root-kit
  - *root* user *nix systems
  - *kit* set of tools to operate/execute commands
- Rootkit is piece or collection of software
  - Designed to enable access where it would be otherwise denied
  - Tries to hide("cloak") its presence in system
- Installed after obtaining privileged access
  - Privileged escalation, credentials compromise, physical access…
- Rootkit != exploit (rootkit usually installed after exploit)
- Rootkit is usually accompanied with additional payload
  - Payload does the actual (potentially malicious) work

Protection rings



- Idea: introduce separate runtime levels
  - crash in level X causes issue only in levels >=X
  - Direct support provided by CPU architectures
- Ring 3: unprivileged user programs
- Ring 2/1: device drivers
- Ring 0: kernel programs
- Performance penalty associated with ring switching
  - In practice, only 3 and 0 are commonly used
- Captures only rings/levels starting with OS
  - Levels -1/-2/-3 introduced for layers below OS

Rootkit		Ring level
Managed code rootkits	Ring "3+"	Managed code (runtime, JVM)
User-mode rootkits	Ring 3	User-mode
Kernel rootkits	Ring 1,2	Device drivers
	Ring 0	OS kernel, device drivers
Hypervisory-level rootkits	Ring -1	Hypervisory-level (VT-x, AMD-V)
SMM abuse, bootkits	Ring -2	System Management Mode, BIOS
FW/HW rootkits	Ring -3	Firmware, hardware

Principal ways of detection of rootkits

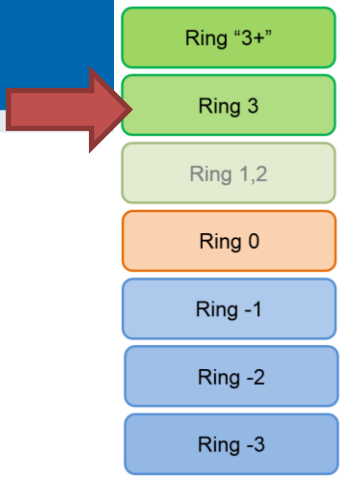
1. Detection running on system, same or higher level
   - Flaws in rootkit cloaking, side-channel
2. Detection running on system, lower level
   - Not controlled by rootkit, cannot cloak itself
3. Detection via (offline) image of system / memory
   - Rootkit is not running => cannot cloak itself

Types of offensive software (wrt OS)

- Alternative taxonomy – how interfaced with system

Category	Interface to system	Location	Examples
Type 0	Existing APIs	Rings 3, 0	Filter driver, Browser helper objects, injected DLL/thread
Type I	Modified static components	Rings 3, 0	Hooking, patching, bootkits
Type II	Modified dynamic components	Rings 3, 0	Direct kernel object modification, Rogue callbacks
Type III	Outside of the system	Rings -1, -2,-3	Rogue hypervisor, firmware mod

J. Rutkowska, Introducing Stealth Malware Taxonomy

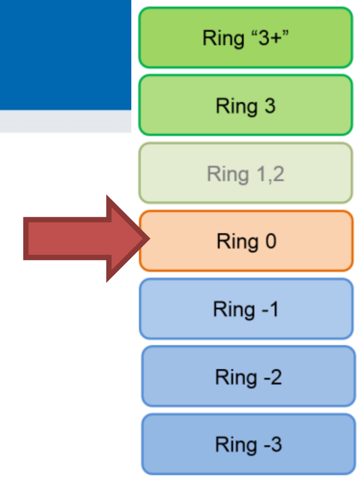


User-mode rootkits (Ring 3)

- Injects payload into other user applications
  - Injection of modified dlls (user app will use different CreateFile)
  - Modification of applications (modification of CreateFile)
- Interception of messages
  - RegisterWindowMessage()
- Function hooking
  - More generic hooks (SetWindowsHookEx()) – window manager
  - User application-specific hooks (plugins, example browser hook)
- File-system filters
  - Detect access to files by user application

Managed code rootkits (MCR) (Ring 3)

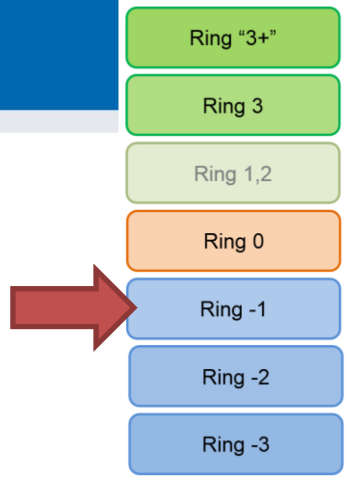
- Ring 3 (level for runtime / VM)
- Targets runtime environments for interpreted code
  - .NET VM, Java VM and Dalvik runtime…
- Large attack surface for MCR
  - Attacking runtime class libraries
  - Attacking JIT compiler
  - Abusing runtime instrumentation features
  - Extending language with malware API
  - Object-oriented malware (inside OO runtime)
- E. Metula: Managed Code Rootkits (Syngress)



Kernel-mode rootkits (Ring 0)

- Runs with highest system privileges
  - Usually device drivers and loadable modules
  - Device drivers in MS Windows
  - Loadable kernel modules in Linux
- Direct kernel object manipulation
  - Data structures like list of processes…
  - System Service Descriptor Table (SSDT) hook
  - System call table hook
- Operating system may require mandatory drivers signing
  - More difficult to insert malicious driver
  - Still possible (compromised private keys: Stuxnet & Realtek's keys)
  - Rootkits below OS level

ROOTKITS BELOW OS LEVEL

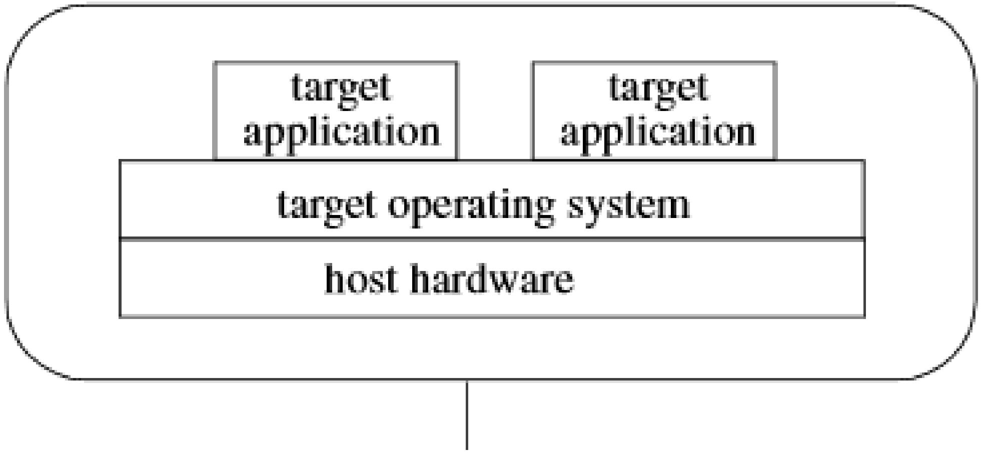


Hypervisory-level rootkits (Ring -1)

- Virtual-machine based rootkit (VMBR)
  - Type II hypervisors (VM under ordinary OS host)
- Based on CPU hardware virtualization features
  - Intel VT or AMD-V
- Rootkit hosts original target as virtual machine
  - And intercepts all relevant hardware calls
- Examples: SubVirt, BluePill (AMD-V, Intel VT-x)

Hypervisory-level rootkits (Ring -1)

Before infection

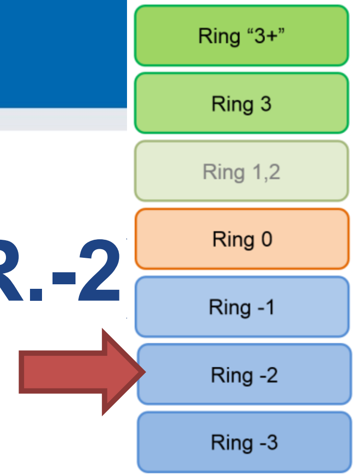


King et al: SubVirt: Implementing malware with virtual machines

Defense against hypervisory-level rootkits

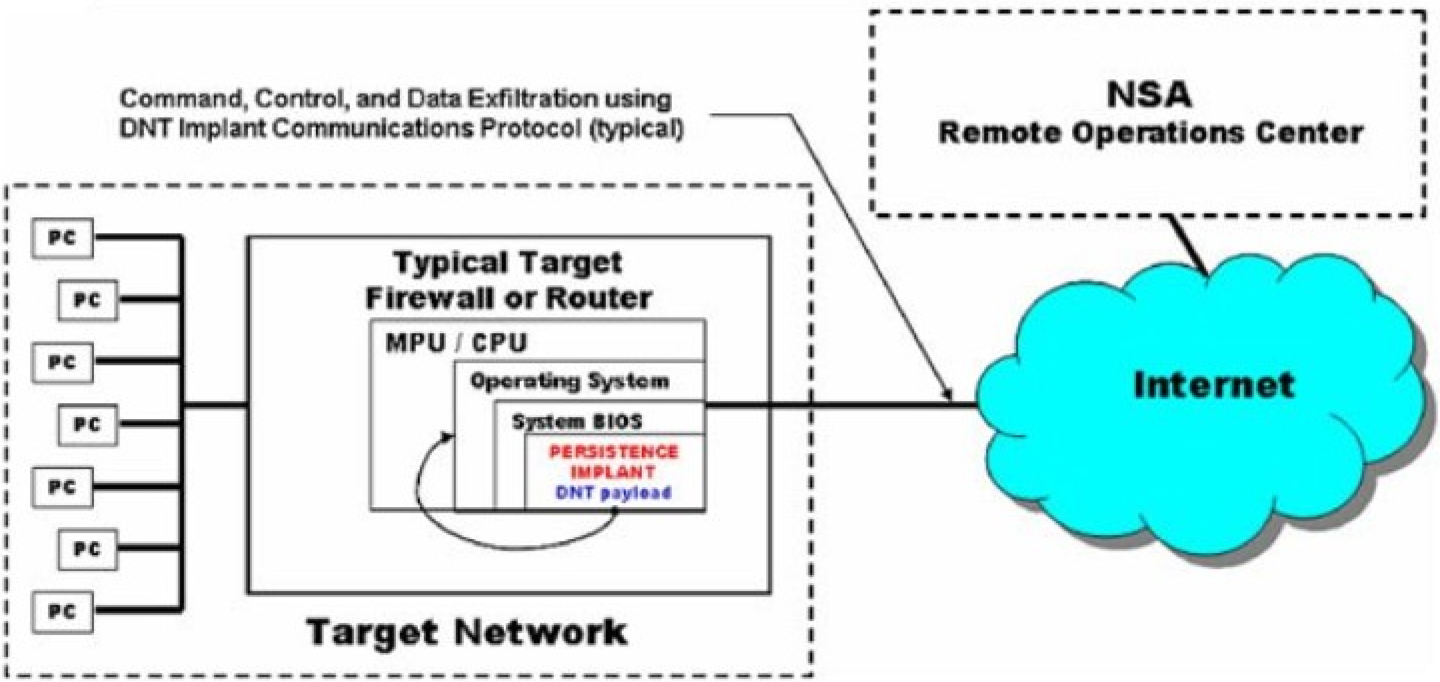
- Run detection/prevention on lower level
- Detect by timing differences of operations
  - System is emulated => side-channel info (timings…)
- Read and analyze HDD physical memory
  - After physical removal from (infected) computer
- Boot from safe medium (CD, USB, network boot)
  - inspect before VMBR loads
  - But VMBR can emulate shutdown / reboot
    - Physical power unplug recommended
- Trusted boot (based on TPM, lecture 13)

System Management Mode abuse (R.-2



- System Management Mode (SMM)
  - x86 feature since Intel 386, all normal execution is suspended
  - Used for power management, memory errors, hardware-assisted debugger…
  - High-privilege mode (Ring -2)
- SMM entered via system management interrupt (SMI)
  - System cannot override or disable the SMI
- Target for rootkits
  - Modify memory, loaders, MBR…

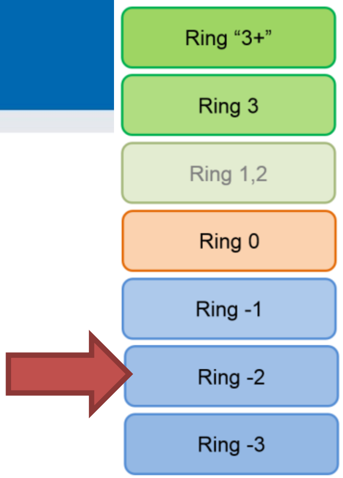
SMM Example: SOUFFLETROUGH implant



(TS//SI//REL) SOUFFLETROUGH Persistence Implant Concept of Operations

- https://en.wikipedia.org/wiki/NSA_ANT_catalog
- http://leaksource.info/2013/12/30/nsas-ant-division-catalog-of-exploits-for-nearly-every-major-software-hardware-firmware/

Bootkit rootkits (Ring -2)



- Bootkit = Rootkit + Boot capability
- Infect startup code
  - Master Boot Record (MBR)
  - Volume Boot Record (VBR)
  - Boot sector, BIOS routines…
- “Evil maid” attack
  - Can be used to attack full disk encryption
  - Assumption: user will left device physically unattended
  - Legitimate bootloader replaced (+ key capture)

Full-disk encryption compromise

1. Full-disk encryption used to encrypt all data
2. Laptop powered down to prevent Coldboot or FireWire-based attacks (read key from memory)
3. Laptop left unattended ("Evil maid" enters)
   - USB used to read part of first sector of disk
   - If TrueCrypt/Bitlocker loader, then insert malicious bootloader

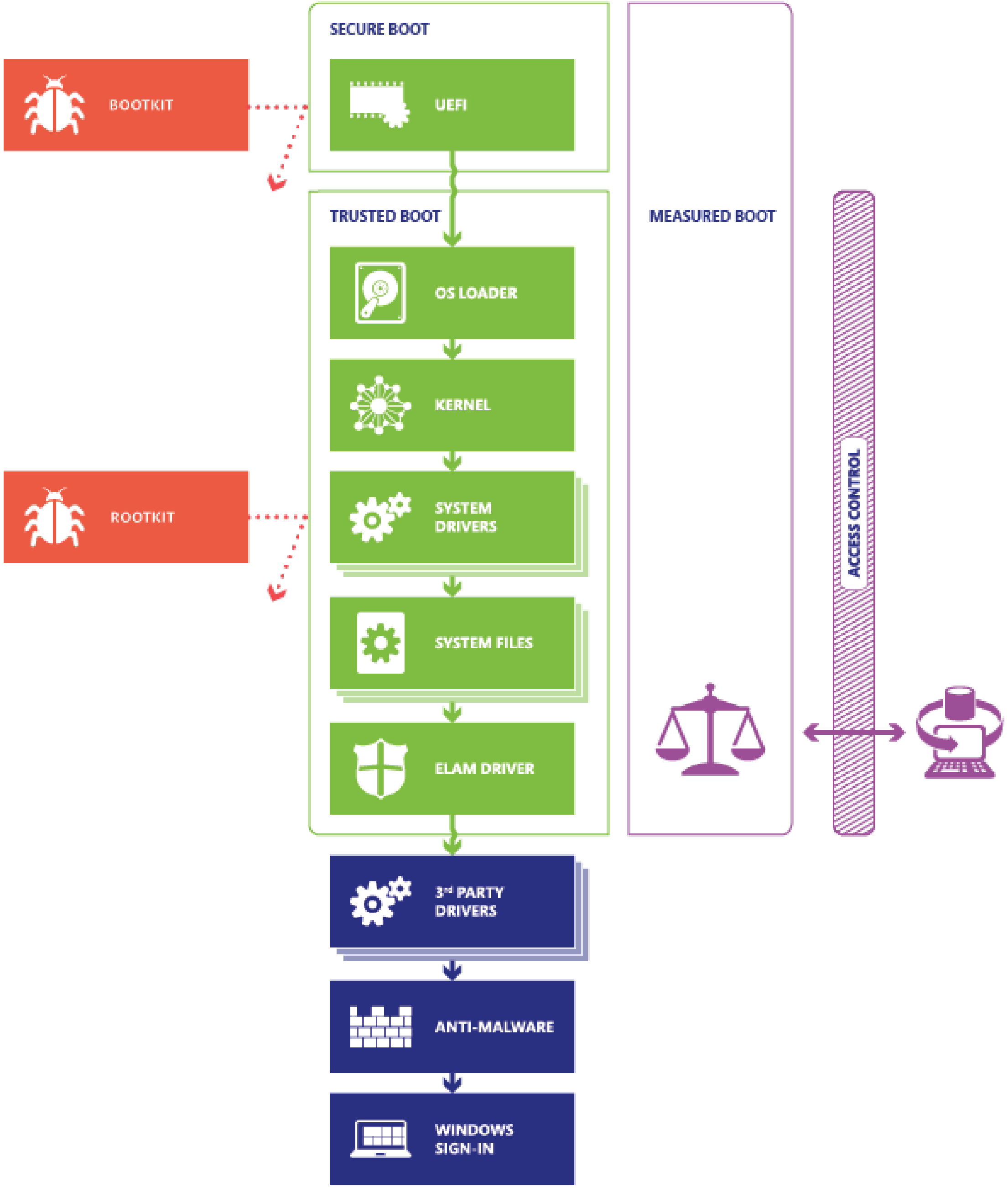
4. User is prompted with forged bootloader
   - Password is stored

- How to transfer saved password / data to attacker?
  - Second visit of Evil maid

http://theinvisiblethings.blogspot.co.uk/2009/10/evil-maid-goes-after-truecrypt.html

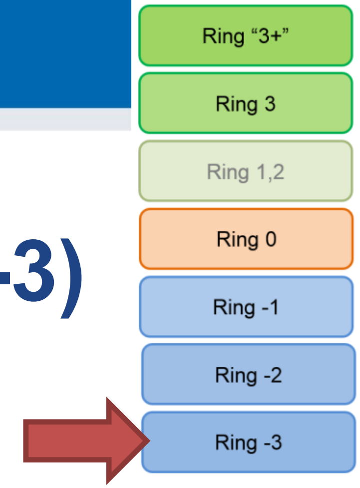
Bootkit defenses

- Prevention of physical access
  - Problematic for portable devices
- Trusted boot (static vs. dynamic root of trust)
  - More in Lecture 13 (Trusted boot)
  - But bootloader must authenticate itself to user
    - E.g., present image encrypted by key stored in TPM
    - Before user enters its password
- Defense by external verification of bootloader integrity
  - verify relevant unencrypted parts of disk (external USB)



http://technet.microsoft.com/en-US/windows/dn168167.aspx

Firmware / hardware rootkits (Ring -3)



- Persistent malware image in hardware
  - Network card, router, hard drive…
- Can run even after removal of device from target computer
  - Once device is powered again

LEGITIMATE USES

Legitimate uses of rootkits

- To whom is legitimacy measured?

- Hide true nature of network “honeypots”
- Protection of AV software against termination
- Anti-theft protections
- Digital rights management

Sony BMG Extended copy protection

- Rootkit developed for (and approved) by Sony
  - Intended to limit possibility for disk copy
  - Users were not notified (silently installed after CD insert)
  - Digital rights management for Sony
  - To hide itself, any file starting with *sys* was hidden
- Detected by M. Russinovich's RootkitRevealer
  - After public disclose, other malware started to hide itself by naming its files as *sys* (user was already "infected")
- Sony released patch for removal (web-based uninstaller)
  - Even more serious flaw introduced (any visited page can install and run program)
  - Resulted in class-action lawsuit against Sony BMG

REVERSE ENGINEERING

Reverse engineering

- A process of knowledge or design extraction from final product (usually man-made)
- Engineering:
  - Mental model → blueprints/source-code → product/binary
- Reverse engineering (back engineering):
  - From product back to knowledge or design
  - Blueprints/source-code might be also recreated
- Not necessary/possible to perfectly recreate design
  - Engineering might be loose transformation
  - Back engineering might not be perfect/complete

Reverse engineering is general process
We will focus on software binaries only

Reverse engineering - legal issues

- Reverse engineering is legal when
  - Own binary without documentation
  - Anti-virus research, Forensics…
  - Interoperability, Fair use, education
- Problem with some copyright laws
  - not only selling circumvented content, but also attempt to circumvent is illegal
- EFF Coders' Rights Project Reverse Engineering FAQ
  - Legal doctrines, Risky aspects, Selected decisions
  - https://www.eff.org/issues/coders/reverse-engineering-faq

How to start reverse engineering

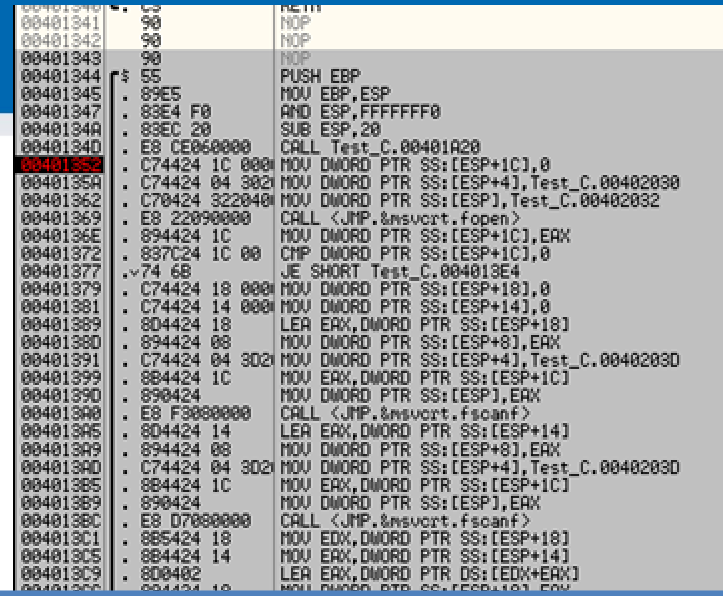
1. Learn basic concepts (compilers, memory, OS…)
2. See how source-code translates into binary
3. Try tools on simple examples (own code, tuts)
4. Utilize other knowledge (communication logs…)
5. Have fun! ☺

Basics

- Debugger vs. debugger with binary modification capabilities
  - E.g., Visual Studio vs. OllyDbg
- Disassembler vs. debugger
  - Static vs. dynamic code analysis
- Disassembler vs. decompiler
  - Native code → assembler → source code
- Native code vs. bytecode
  - Different instruction set, different execution model
- Registry-based vs. stack-based execution

Mixed source code/assembler in IDE

- Most current IDE supports mixed source code/assembler instructions mode (Visual Studio, QT Creator...)
  - Mode is usually available only during a debugging
  - Write simple code (e.g., if then else condition), insert breakpoint and start debugging
- Switch to mixed mode
  - Visual Studio→RClick→Go to disassembly
  - QTCreator→Debug→Operate by Instruction
- Easy way to learn how particular source code is translated into assembler code




```c
#include <stdio.h>
int main() {
    FILE* file = NULL;
    file = fopen("values.txt", "r");

    if (file) {
        int value1 = 0;
        int value2 = 0;
        fscanf(file, "%d", &value1)
        fscanf(file, "%d", &value2)

        value1 = value1 + value2;

        printf("Result: %d", value1)
    }
    fclose(file);
}
```

Original C source code

```
Dump of assembler code for function main:
       2        int main() {
0x00401344  <+0>:          push   %ebp
0x00401345  <+1>:          mov    %esp,%ebp
0x00401347  <+3>:          and    $0xfffffff0,%esp
0x0040134a  <+6>:          sub    $0x20,%esp
0x0040134d  <+9>:          call   0x401a20 <__main>

       3            FILE* file = NULL;
0x00401352  <+14>:         movl   $0x0,0x1c(%esp)

       4            file = fopen("values.txt", "r");
0x0040135a  <+22>:         movl   $0x402030,0x4(%esp)
0x00401362  <+30>:         movl   $0x402032,(%esp)
0x00401369  <+37>:         call   0x401c90 <fopen>
0x0040136e  <+42>:         mov    %eax,0x1c(%esp)

            ...

      17        }
0x004013f5  <+177>:        leave
0x004013f6  <+178>:        ret
       End of assembler dump.
```

Most common instructions/structures

- Most common ASM instructions
  - Load/Store from to registers: MOV, LEA
  - Arithmetic: ADD, INC…
  - Relational: CMP, TEST
  - Jumps: JMP, J*
  - Functions: CALL, RET
- Example of typical structures (C→ ASM)
  - Conditional jump, `for` loop, function call…
  - Familiarize via mixed source code/assembler in IDE
  - Be aware of debug/release differences

Compilation to bytecode (Java, C#)

- Source code compiled into intermediate bytecode
  - Java bytecode,.NET CLI ...
- Intermediate code interpreted by virtual machine
- Just-in-time compilation
  - Intermediate code is compiled by VM into native code
  - Improve performance significantly
  - Relevant for dynamic analysis, not for static analysis
- Usually easier to understand then assembler code

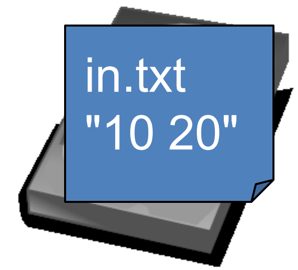
REGISTRY VS. STACK-BASED EXECUTION

Registry-based execution

1. Values loaded (`mov`) from RAM to CPU registers
2. CPU operation (`add, inc, test`...) is executed
3. Resulting value is stored back (`mov`) to RAM

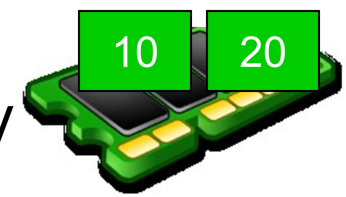
- Name of the registers
  - EAX 32bit, AX 16bit, AH/AL 8bit
  - EIP ... next address to execute (instruction pointer)
  - EBX ... usually loop counter
- Registers
  - Z – zero flag, C – carry flag, S – sign flag…

Add two numbers from file (HDD)

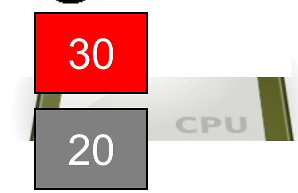


`value = value + value2;`

1. Read values from HDD into RAM memory
   - `fscanf(file, "%d", &value);`



2. Move value from RAM memory to CPU registry
   - `MOV 0x48(%esp),%eax`
   - `MOV 0x44(%esp),%edx`



3. Execute CPU instruction (e.g., ADD)
   - `ADD %edx,%eax`

4. Transfer result from CPU register to RAM memo
   - `MOV %eax, 0x48(%esp)`
5. Save result from RAM memory to file
   - `fprintf(file, "%d", value);`

Stack-based execution

- Bytecode contains sequence of operations
- Bytecode contains constants
- All intermediate values stored on stack
- Interpret:

1. Reads next operation from bytecode
2. Pop operand(s) for next operation from top of stack
3. Executes operation
4. Push result of operation on top of stack

- No registers are used
  - all operands for current operation at the top of the stack

CROCS

Example: JavaCard bytecode

```
// ENCRYPT INCOMING BUFFER
  void Encrypt(APDU apdu) {
      byte[]      apdubuf = apdu.getBuffer();
      short       dataLen = apdu.setIncomingAndReceive();
      short       i;

      // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
      if ((dataLen % 8) != 0)
         ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

      // ENCRYPT INCOMING BUFFER
      m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen,
                              m_ramArray, (short) 0);

      // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
      Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf,
                              ISO7816.OFFSET_CDATA, dataLen);

      // SEND OUTGOING BUFFER
      apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
  }
```

Original JavaCard source code

```
.method  Encrypt(Ljavacard/framew
 .stack 6;
 .locals 3;
 .descriptor    Ljavacard/framework
L0:     aload_1;
        invokevirtual 30;
        astore_2;
        aload_1;
        invokevirtual 42;
        sstore_3;
        sload_3;
        bspush 8;
        srem;
        ifeq L2;
L1:     sspush 26384;
        invokestatic 41;
        goto L2;
L2:     getfield_a_this 1;
        aload_2;
        sconst_5;
        sload_3;
        getfield_a_this 10;
        sconst_0;
        invokevirtual 43;
        pop;
        getfield_a_this 10;
        sconst_0;
        aload_2;
        sconst_5;
        sload_3;
        invokestatic 44;
        pop;
        aload_1;
        sconst_5;
        sload_3;
        invokevirtual 45;
        return;
}
```

Resulting JavaCard bytecode

Recovering information from binary executables

DISASSEMBLING

Disassembling of native binaries

- Reversing process of compilation
  - Back from native code to ASM
- Compilation/assembly is loose process:
  - Variable/function names
  - Unused structures
  - Performance optimization applied during compilation
- Wide range of native platforms
  - Differences in support and performance of disassemblers
- Bytecode is already on the level of “disassembled” binaries (usually easier to understand)

Structured code vs. sequence of executed ops

- Structured code contains code for all branches
  - runnable binary/bytecode
- Information loss in compiled binary
  - Stripped metadata and debugging symbols
  - Compiler optimizations
- Sequence of executed instructions only from branches taken
  - E.g., power analysis of smart card

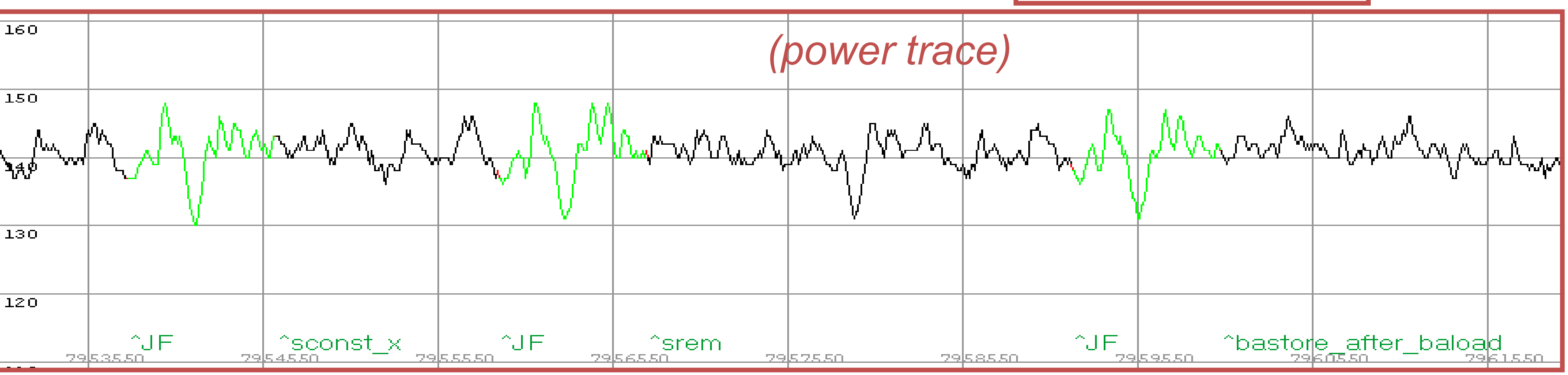
Structured code vs. sequence of executed ops

(source code)
m_ram1[0] = (byte) (m_ram1[0] % 1);

compiler →

(bytecode)
getfield_a_this 0;
sconst_0;
baload;
sconst_1;
srem;
bastore;

oscilloscope ↓



Bytecode reconstruction →

(partial bytecode)
…; sconst_???; baload; sconst_???; srem; bastore;…

Tool: OllyDbg

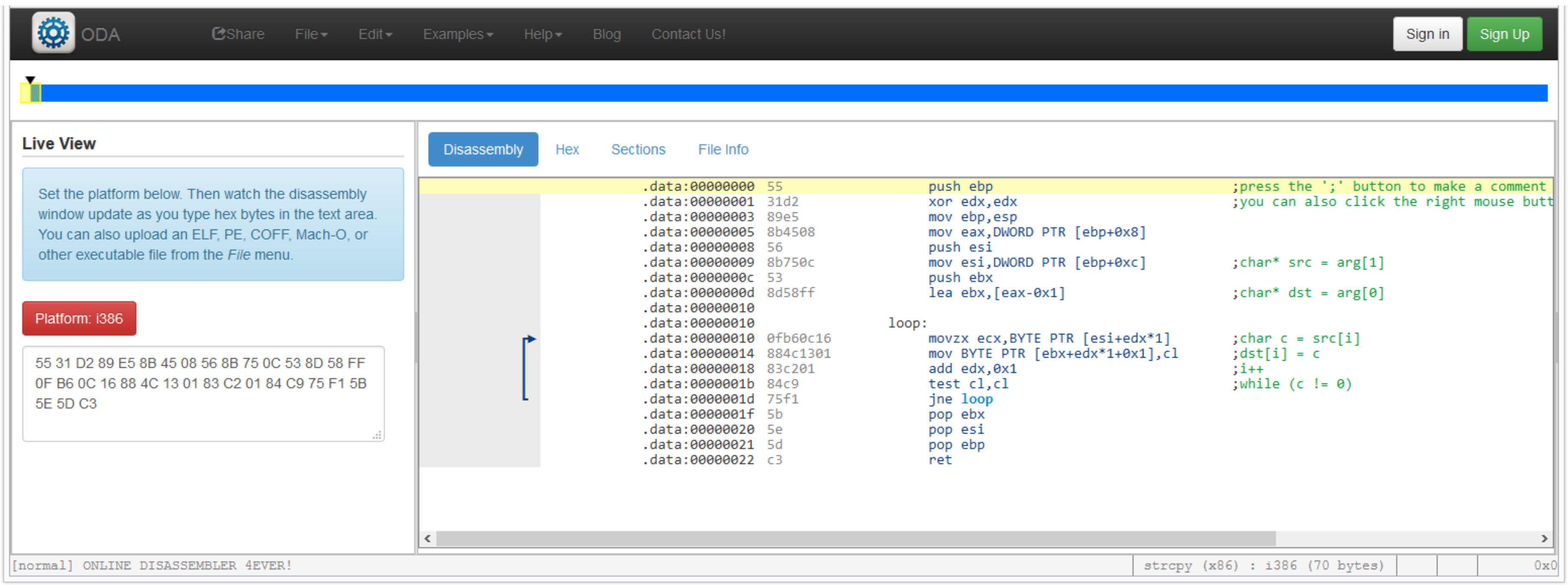
- Free disassembler and binary debugger
  - Works with Windows 32b binaries only
  - OllyDbg 64b version in development
- Easy to start with, many tutorials
- Designed to make changes in binary easy
  - Change of jumps/data (valid PE is recreated)
- http://www.ollydbg.de/

Tool: IDA Pro

- Interactive Disassembler is legendary full-fledged disassembler with ability to disassemble many different platforms
- Free version available for non-commercial uses
  - http://www.hex-rays.com/idapro/idadownfreeware.htm
- Free version disassemble only Windows binaries
- Very nice visualization and debugger feature (similar as OllyDbg)

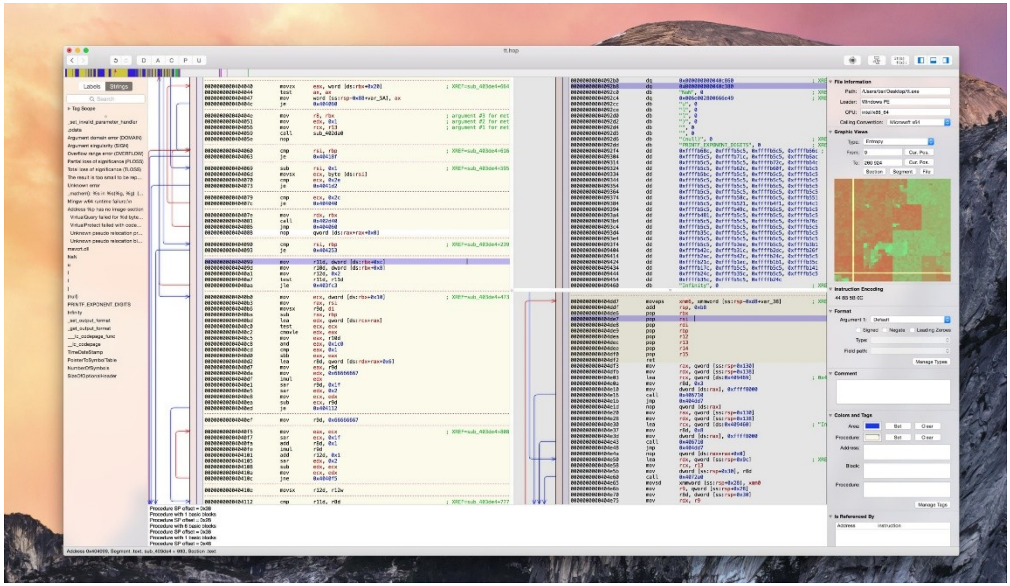
Tool: Online disassembler (ODA)

- https://www.onlinedisassembler.com/odaweb/



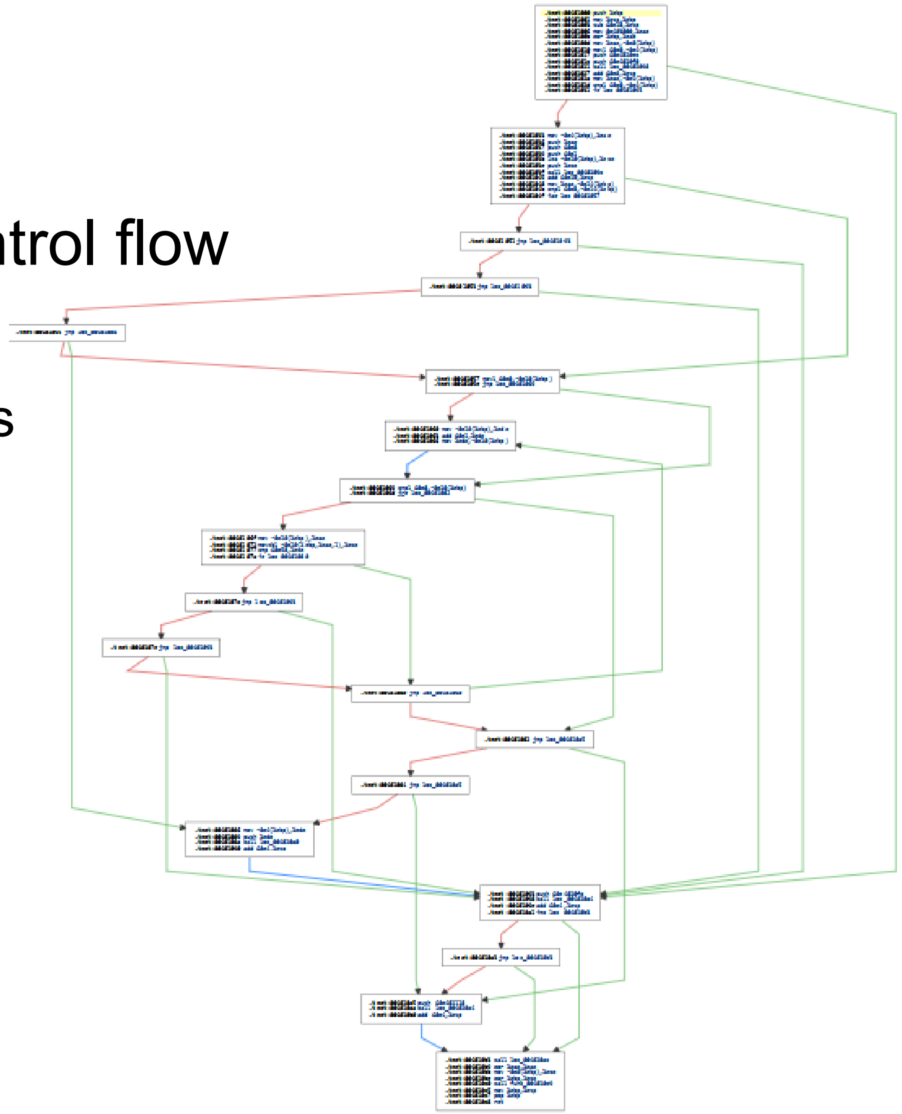
Tool: Hopper diassembler and debugger

- Linux and OS X reverse engineering tool
  - Older version supported Windows, but not anymore
- http://www.hopperapp.com
- Additional support for on Objective-C



Control flow graph

- Graph representation of control flow
- Separated functions/blocks
  - connection by jump instructions



Decompilation

- Native code decompilation
  - Decompiler produces source code from binary/ASM/bytecode code
  - Decompiler needs to do disassembling first and then try to create code that will in turn produce binary code you have at the beginning
  - Resulting code will NOT contain information removed during compilation (comments, function names, formatting...)
- Bytecode decompilation
  - usually much easier (more information preserved)
  - Mapping between source code and bytecode is less ambiguous
  - Compilation of decompiled bytecode produces similar bytecode

Decompiler tools

- C/C++
  - IDA
  - REC Studio 4.0, http://www.backerstreet.com/rec/rec.htm
  - Retargetable Decompiler, https://retdec.com/
- Java bytecode
  - DJ Java Decompiler, http://neshkov.com/dj.html
  - Java Decompiler, http://jd.benow.ca/
- .Net bytecode
  - dotPeek, https://www.jetbrains.com/decompiler/
  - ILSpy, http://ilspy.net/

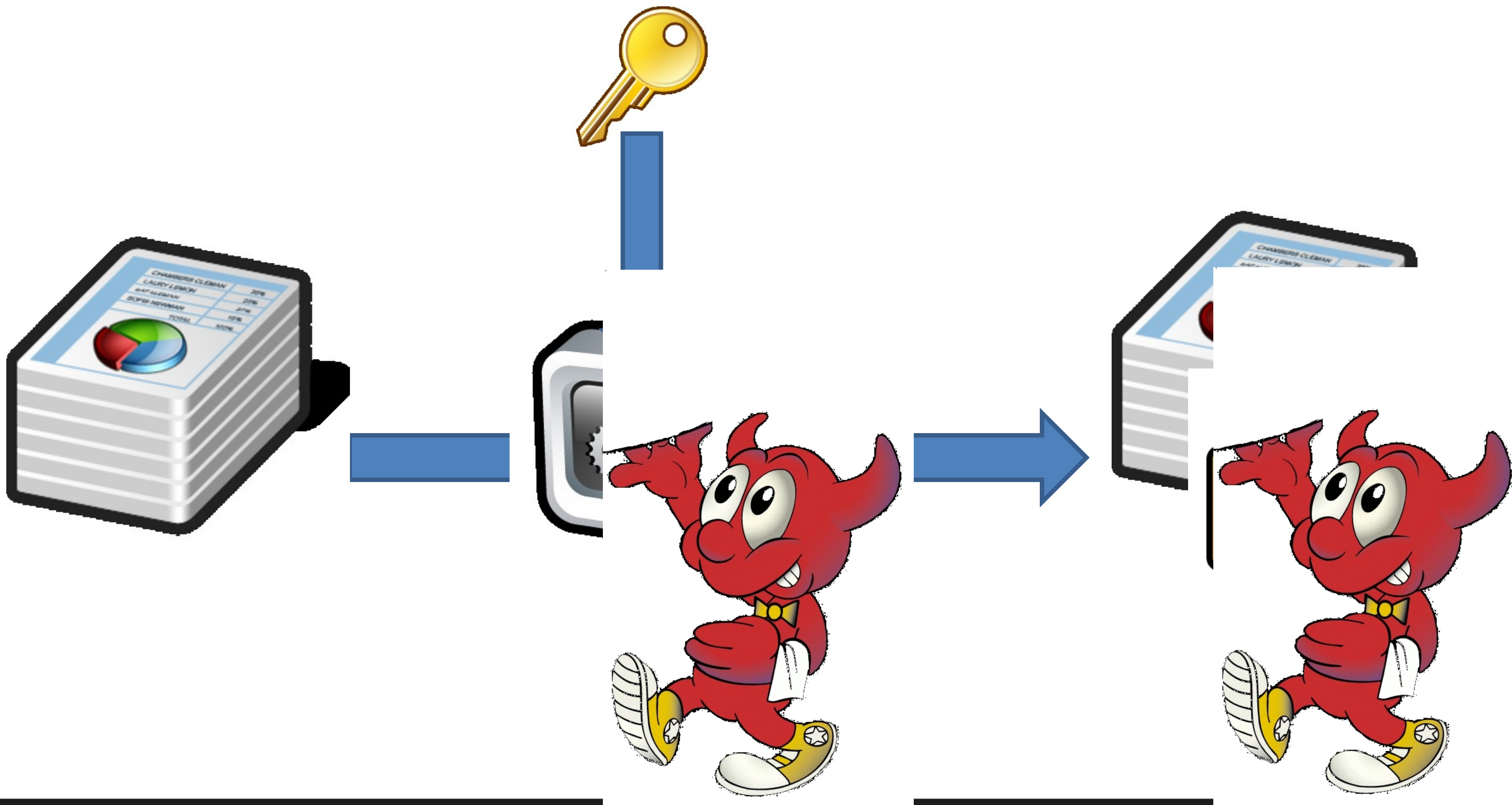
Resources

- Reverse Engineering for Beginners
  - http://beginners.re/Reverse_Engineering_for_Beginners-en.pdf
  - Great resource, many examples, tutorials
- Tutorials for You: http://www.tuts4you.com
- The Reverse Code Engineering Community: http://www.reverse-engineering.net/
- Disassembling tutorial http://www.codeproject.com/KB/cpp/reversedisasm.aspx

Protections Against Reverse Engineering

HOW TO PROTECT

Standard vs. whitebox attacker model (symmetric crypto example)



Classical obfuscation and its limits

- Time-limited protection
- Obfuscation is mostly based on obscurity
  - add bogus jumps
  - reorder related memory blocks
  - transform code into equivalent one, but less readable
  - pack binary into randomized virtual machine...
- Barak’s (im)possibility result (2001)
  - family of functions that will always leak some information
  - but practical implementation may exists for others
- Cannetti et. al. positive results for point functions
- Goldwasser et. al. negative result auxiliary result

Computation with Encrypted Data and Encrypted Function

CEF&CED

CEF

- Computation with Encrypted Function (CEF)
  - A provides function F in form of P(F)
  - P can be executed on B's machine with B's data D as P(D)
  - B will not learn function F during computation

CED

- Computation with Encrypted Data (CED)
  - B provides encrypted data D as E(D) to A
  - A is able to compute its F as F(E(D)) to produce E(F(D))
  - A will not learn D

CED via homomorphism

1. Convert your function into circuit with additions (**`xor`**) and multiplications (**`and`**) only
2. Compute addition and/or multiplication “securely”
   – an attacker can compute E(D1+D2) = E(D1)+E(D2)
   – but will learn neither D1 nor D2
3. Execute whole circuit over encrypted data

- Partial homomorphic scheme
  – either addition or multiplication is possible, but not both
- Fully homomorphic scheme
  – both addition and multiplication (unlimited)

Partial homomorphic schemes

- Example with RSA (*multiplication*)
  - $E(d_1).E(d_2) = d_1^e \,.\, d_2^e \bmod m = (d_1 d_2)^e \bmod m = E(d_1 d_2)$
- Example Goldwasser-Micali (*addition*)
  - $E(d_1).E(d_2) = x^{d1} r_1^2 \cdot X^{d2} r_2^2 = x^{d1+d2} (r_1 r_2)^2 = E(d_1 \oplus d_2)$
- Limited to polynomial and rational functions
- Limited to only one type of operation (*mult* or *add*)
  - or one type and very limited number of other type
- Slow – based on modular mult or exponentiation
  - every operation equivalent to whole RSA operation

Fully homomorphic scheme - usages

- Outsourced cloud computing and storage
  - FHE search, Private Database Queries
  - protection of the query content
- Secure voting protocols
  - yes/no vote, resulting decision
- Protection of proprietary info - MRI machines
  - expensive algorithm analyzing MR data, HW protected
  - central processing restricted due to private patient's data
- …

Fully homomorphic scheme (FHE)

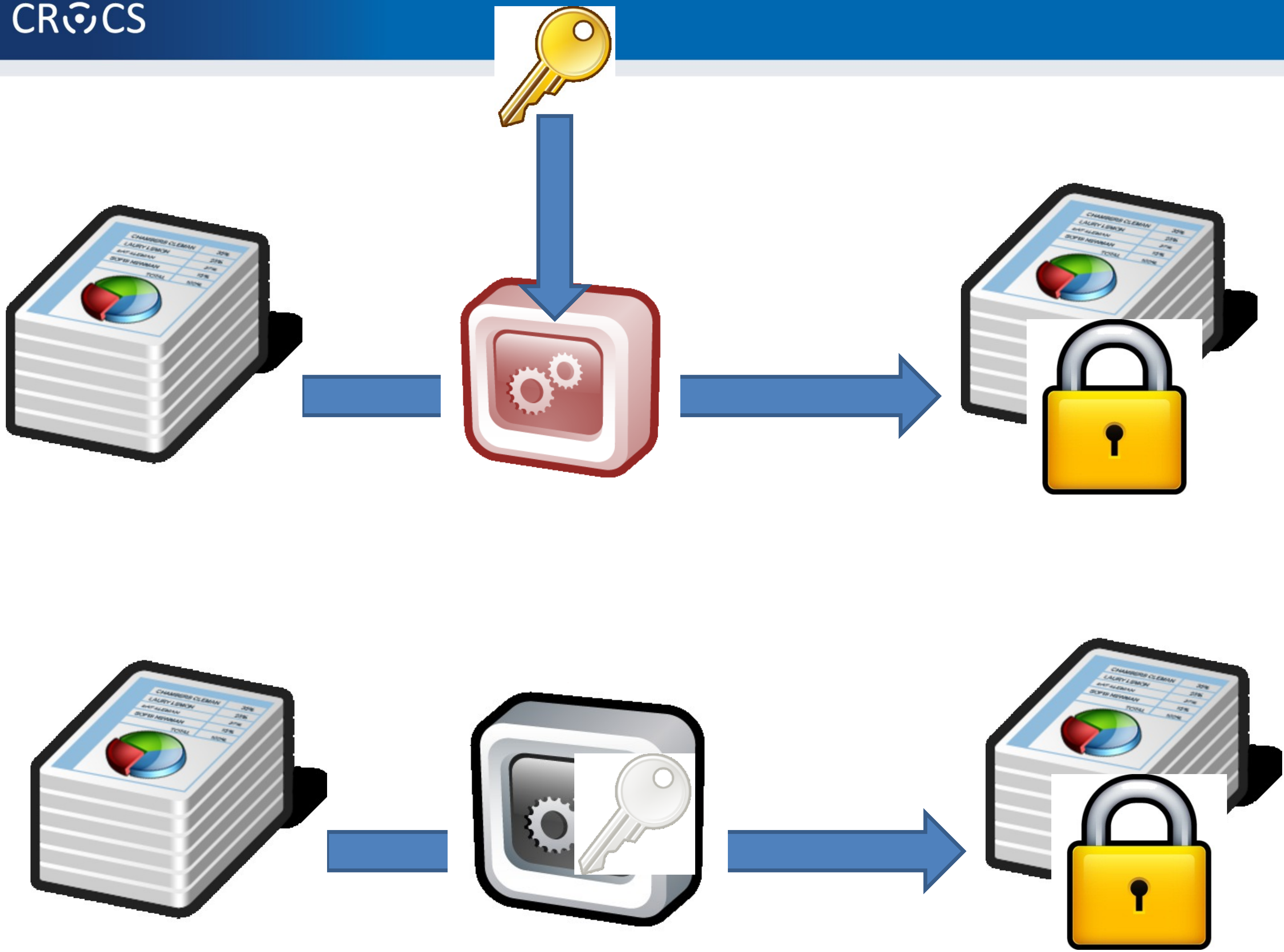
- Holy grail - idea proposed in 1978 (Rivest et al.)
  - both addition and multiplication securely
- But no scheme until 2009 (Gentry)!
  - based on lattices over integers
  - noisy FHE usable only for few operations
  - combined with repair operation (enable to use it for more again)

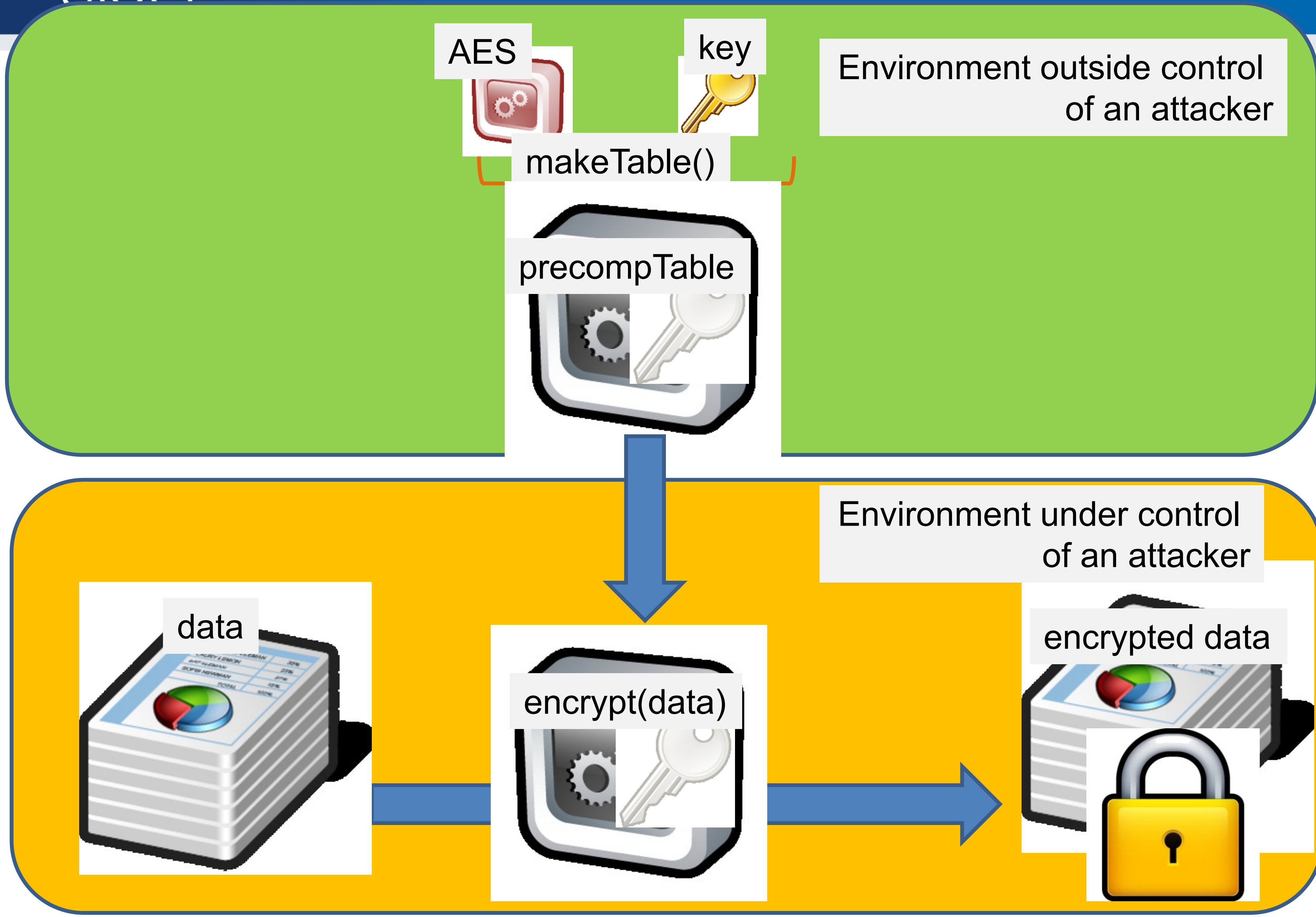
Fully homomorphic scheme - practicality

- Not very practical (yet ☺) (Gentry, 2009)
  - 2.7GB key & 2h computation for every repair operation
  - repair needed every ~10 multiplication
- FHE-AES implementation (Gentry, 2012)
  - standard PC ⇒ 37 minutes/block (but 256GB RAM)
- Gentry-Halevi FHE accelerated in HW (2014)
  - GPU / ASICS, many blocks in parallel => 5 minutes/block
- Replacing AES with other cipher (Simon) (2014)
  - 2 seconds/block
- Very active research area!

White-box attack resistant cryptography

- Problem limited from every cipher to symmetric cryptography cipher only
  - protects used cryptographic key (and data)
- Special implementation fully compatible with standard AES/DES… 2002 (Chow et al.)
  - series of lookups into pre-computed tables
- Implementation of AES which takes only data
  - key is already embedded inside
  - hard for an attacker to extract embedded key
  - Distinction between key and implementation of algorithm (AES) is removed




AES
key
Environment outside control of an attacker
makeTable()
precompTable
Environment under control of an attacker
data
encrypt(data)
encrypted data

WBACR AES - pros

- Practically usable
  - implementation size ~800KB (tables)
  - speed ~MBs/sec (~6.5MB/s vs. 220MB/s)
- Hard to extract embedded key
  - Complexity semi-formally guaranteed
  - (if the scheme is secure)
- One can simulate asymmetric cryptography!
  - implementation contains only encryption part of AES
  - until attacker extracts key, decryption is not possible

WBACR AES - cons

- Implementation can be used as oracle (black box)
  - attacker can supply inputs and obtain outputs
  - even if she cannot extract the key
  - (can be partially solved by I/O encodings)
- Problem of secure input/output
  - protected is only AES, not code around
- Key is fixed and cannot be easily changed
- Successful cryptanalysis for several schemes
  - several former schemes broken
  - new techniques proposed

Whitebox transform IS used in the wild

- Proprietary DRM systems
  – details are usually not published
  – AES-based functions, keyed hash functions, RSA, ECC...
  – interconnection with surrounding code
- Chow at al. (2002) proposal made at Cloakware
  – firmware protection solution
- Apple’s FairPlay & Brahms attack
  - http://whiteboxcrypto.com/files/2012_MISC_DRM.pdf
- ...